CT421 Artificial Intelligence

Uninformed Search

Search

- Many problems can be re-cast or viewed as a search problem.
- Consider designing an algorithm to solve a suduko puzzle.
- In every step, we are effectively searching for a move (an action) that takes us to a correct legal state. To complete the game, we are iteratively searching for an action that brings us to legal board and so forth until complete.

Other Examples:

- Searching for a path in a maze
- Word ladders
- Chess/Checkers

Suduko

			3	4	2	1	6	
4			8	1	6			
	1	6	7	5	9			4
		7		8	3	9		
5	8			6			4	3
		4	9	2	5	7		
7					4	8	2	
			2		1			5
	3	2	5		8			

Formalizing the problem statement

- Problem can be in various states
- Start in an initial state
- There are a set of actions available
- Each action changes the state
- Each action has an associated cost
- Want to reach some goal while minimizing cost

More formally

- Set of states S
- Start state $s_0 \in S$
- Set of actions A and action rules $a(s) \rightarrow s^{'}$
- A goal test $g(s) \rightarrow 0, 1$
- Cost function $C(s, a, s^{'}) \rightarrow \mathbb{R}$
- Search can be defined by the 5-tuple (S, s, a, g, C)

Problem Statement

Find a sequence of actions $a_1 \ldots a_n$ and corresponding states $s_0 \ldots s_n$ such that

- $s_0 = s$
- $s_i = a_i(S_{i-1})$
- $g(s_n) = 1$

while minimizing: $\sum_{i=1}^{n} c(a_i)$

Suduko

- Sudoku States: all legal Sudoku boards.
- Start state: a particular, partially filled-in, board.
- Actions: inserting a valid number into the board.
- Goal test: all cells filled and no collisions.
- Cost function: 1 per move.

- We can conceptualise this search as a search tree.
- A node represents a state.
- The edges from a state represent the possible actions from that state. The edge point to the new resulting state from the action.

Important factors of a search tree

- The breadth of the tree (branching factor)
- The depth of the tree
- The minimum solution depth
- Size of the tree $O(b^d)$
- The set of unexplored nodes that are reachable from any currently explored node is known as the *frontier*
- Choosing which node to explore next is the key in search algorithms

Initialise

```
visited = {};
frontier = {s_0};
goal_found = false;
```

```
while !(goal_found)
        node = frontier.next();
        frontier.del(node);
        if(g(node));
            goal_found = true;
        else
            visited.add(node)
            forall child in node.children
                if(not visited.contains(child))
                    frontier.add(child)
```

- The manner in which we expand the node is key to how the search progresses.
- The way in which we implement ($\mathit{frontier}.\mathit{next}()$) determines the type of search.
- Otherwise the basic approach above remains unchanged.

Uninformed Search

- Nothing known (or used) about solutions in the tree.
- Possible approaches?
  - Expand deepest node (depth-first search)
  - Expand closest node (breadth-first search)
- Properties
  - Completeness
  - Optimality
  - Time Complexity (total number of nodes visited)
  - Space Complexity (size of frontier)

Depth First Search

- Space: $O(bd)$
- Time: $O(b^d)$
- Completeness: Only for finite trees.
- Optimality: No.

Breadth First Search

- Space: $O(b^{m+1})$, where m is the depth of the solution
- Time: $O(b^m)$, where m is the depth of the solution in the tree
- Completeness: Yes.
- Optimality: Yes (assuming constant costs)

Introduction

- DFS: good regarding memory cost; however, suboptimal solution.
- BFS: optimal solution, but expensive memory cost.

Iterative Deepening Search

- Iterative Deepening attempts to overcome some of the issues of both of the above.
- Run DFS to a fixed depth z.
- Start at $d = 1$ If no solution, increment d and rerun.

Efficiency?

- Low memory requirements (equal to DFS).
- Not many more nodes expanded than BFS.
- Note the leaf level will have more nodes than the previous layers

- Let's consider the case where the costs are not uniform; thus far we have assumed each edge has a fixed cost.
- Neither DFS or BFS are guaranteed to find the least-cost path, in the case where action costs are not uniform.
- Approach: chose the one with lowest cost?

- Order the nodes in the frontier by cost-so-far (Cost of the path from the start state to the current node)
- Explore next the node with the smallest cost-so-far
- Give the optimal solution
- Complete solution (given all positive costs)

Informed Search

- So far, we have assumed we know nothing about the search space? What should we do if we know something about the space?

- We know the cost of getting to the current node
- Remaining cost of finding solution: cost from current node to goal state
- Total cost: Cost of getting from start to current node + cost of getting from current node to goal state

Approach

- Use an heuristic h(s) to estimate the remaining cost
- h(s) = 0 if s is a goal.
- Problem specific

A* algorithm

- Let $g(s)$ be the cost of the path so far
- This algorithm expands the node s to minimise $g(s) + h(s)$
- Manage frontier nodes as priority queue.
- If h never overestimates the cost, the algorithm will find the optimal solution.

Heuristics

- Fast to compute.
- Close to real costs.

Adversarial Search

Typical Game Setting

- 2 player
- Alternating
- Zero-sum: Gain for one loss for another.
- Perfect information

“Solved” Games

- A game is solved if an optimal strategy is known.
- Strong solved: all positions.
- Weakly solved: some (start) positions.

- Set of possible states
- Start state
- Set of actions
- End states (many)
- Objective function
- Control over actions alternates

Minimax Algorithm

- Compute value for each node, going backwards from the end-nodes.
- Max (min) player: select action to maximize (minimize) return.
- Assumes perfect play, worst case.
- For optimal play, require the agent to evaluate the whole tree

Issues to consider

- Noise/randomness
- Efficiency - size of tree
- Many game trees too deep
- Many game trees too broad

Alpha Beta Pruning

- A means to reduce the search space.
- Can prune sibling nodes based on previously found values.
- Maintain the current maximum (for player 1) and current minimum (for player 2)
- Allows us to discard whole subtrees

- In reality, for many search scenarios in games, even with alpha beta pruning, the space is much too large to get to all end states.
- Instead, we use an evaluation function - effectively an heuristic to estimate the value of a state (probability of win/loss)
- Run search to fixed depth; evaluate all states at that depth
- Perform look ahead from best states to another fixed depth.

Frame Title

Horizon Effects

- What if something interesting/unusual/unexpected occurs at horizon + 1?
- How do you identify?
- When to generate and explore more nodes?
- several algorithms developed to take this into account
- Deceptive problems?